Tools for architecture based autonomic systems

Noel De Palma[1], Konstantin Popov[2], Nikos Parlavantzas[1], Per Brand[2],Vladimir Vlassov[3]
[1]*INRIA, Grenoble, France,* [2]*SICS, Stockholm, Sweden,* [3]*KTH, Stockholm, Sweden*
noel.depalma@inrialpes.fr, kost@sics.se, nikolaos.parlavantzas@inrialpes.fr,
perbrand@sics.se, vladv@kth.se,

Abstract

Recent years have seen a growing interest in autonomic computing, an approach to providing systems with self managing properties [1]. Autonomic computing aims to address the increasing complexity of the administration of large systems. The contribution of this paper is to provide a generic tool to ease the development of autonomic managers. Using this tool, an administrator provides a set of alternative architectures and specifies conditions that are used by autonomic managers to update architectures at runtime. Software changes are computed as architectural differences in terms of component model artifacts (components, attributes, bindings, etc.). These differences are then used to migrate into the next architecture by reconfiguring only the required part of the running system.

1. Introduction

Our computing environments are increasingly sophisticated: they involve numerous, complex software systems that cooperate in potentially distributed environments. These systems are developed with heterogeneous programming models and typically expose proprietary configuration facilities. As a result, the administration of these systems (installation, configuration, tuning, repair…) is becoming increasingly costly in terms of human, hardware, and software resources. A promising approach to reducing the costs of administration is to build autonomic systems. Autonomic systems are able to "reason" about their own properties and to control their own behavior at run time. For this purpose, they embed autonomic managers which implement control loops that regulate the managed system. The managed system can be a single hardware or software element, or it may be a complex system, such as a cluster of machines, or a distributed middleware infrastructure. The autonomic manager interacts with the managed system through two types of elements: sensors to watch the state of the system, and actuators to reconfigure it. Importantly, the autonomic manager also embeds knowledge of the system, which is used as a basis for various management functions.

In the architecture-based management approach [2], the knowledge of the system is a formal or semi-formal description of the organization of the managed system. This description is typically based on components and bindings. The environment of the applications (e.g., hardware resources, system services) may be described in the same way. When an event occurs in the managed system and triggers an autonomic response, the manager first analyses the event, choose a reconfiguration plan to fix the problem and then executes the plan using the system's actuators to change the managed system accordingly. The manager's job can be partially seen as a function that updates the architecture of the system from a given running configuration to a target configuration. In other words, the system is dynamically updated from a component structure to another. The contribution of this paper is to provide a generic tool to ease the development of autonomic managers. Using this tool, an administrator provides a set of alternative architectures and specifies conditions that are used by autonomic managers to update the architecture at runtime. Software changes are computed as architectural differences by an architecture diff algorithm in terms of component model artifacts (components, attributes, bindings...). These differences are then used by a change manager to reconfigure only the required part of the running system to update into the target architecture. As a first step, this work focuses on distributed applications running on a local area network (e.g., a cluster) and under the control of a single autonomic manager.

This paper is organized as follows. Section 2 describes our main requirements, and Section 3 discusses related work. Section 4 then presents our design principles, and Section 5 provides an overview of our architecture. Sections 6, 7 and 8 give more details about the used component model, the diff algorithm and change management. Finally, Section 9 describes a use case, and Section 10 concludes the paper.

2. Requirements

Our approach is driven by the following requirements:

- **Architecture-based management**: the architecture-based approach requires (i) providing meta-data that describe all the alternative architectures of the system, and (ii) providing fine-grained operations to monitor and to reconfigure the architecture at runtime.

- **Architecture comparison**: our autonomic manager requires a comparison algorithm, able to automatically compute the differences between the current runtime architecture of a system and a target architecture. From these differences, the manager should produce a reconfiguration script to dynamically update the current architecture into the target one.

- **Consistent change management**: we require a change manager, able to interpret the reconfiguration script generated by the architecture comparison. Importantly, this manager needs to ensure the system's consistency during reconfiguration.

3. Related work

Much work has focused on the problem of detecting changes between documents, both flat and structured documents. In the context of flat documents, the GNU diff tool supports comparing plain files and reporting their difference; the tool is notably used by `cvs` [13]. Based on the Longest Common Subsequence algorithm, the diff tool is inadequate to compare structured data as it does not understand hierarchical structure information. In the context of structured documents, document comparison often relies on tree distance algorithms. A good example is Valiente's algorithm [12], based on bottom up mapping of the rooted trees given by the largest common forest between them. Many solutions have been provided for XML [4, 10, 11, 14]. Diffx [14] uses tree fragment mapping: it iteratively identifies the largest matching tree fragments between the tree representations of the two versions of the document. Diffx handles differences in both the structure and the content of the two trees. Xdiff [11] is another solution which uses a complete top-down mapping mechanism relying on the node signature when matching the nodes. This signature represents the path from the root to the given node. Xdiff is more suited when the structure of the document remains unmodified.

The comparison problem has also been addressed in the area of object-oriented design. Specifically, in the context of UML, UMLDiff [15] compares the logical view of object-oriented software systems, which concerns classes, the information they may own, the services they can deliver, and the associations and relative organization among them. UMLDiff is a domain-specific structural-differencing algorithm, aware of the UML semantics.

Unlike all this work, we focus on computing differences between running component structures. Moreover, we support dynamic system reconfiguration based on these differences, expressed as sets of reconfiguration operations. Our comparison of component structures is based on a tree comparison algorithm, extended with component model semantics.

4. Design Principles

The requirements described in Section 2 are addressed using the following design principles.

4.1. Architecture-based management

Our first design principle is to use a component-based structure: the managed system is built or, at least, wrapped as a set of components. Components provide control interfaces that are entry points for sensors and actuators, which, together with analyzers and planners, make up the control loop. The component-based structure potentially allows fine-grained dynamic reconfiguration, a prerequisite for autonomic behavior. Our second principle is to enhance the component-based architecture with meta-data. The autonomic manager maintains a meta model of the system's distributed architecture. This meta model is isomorphic to the runtime structure of the system, but adds some meta data concerning the infrastructure as a whole. Specifically, the model contains (i) the application's distributed architecture in terms of component and binding, (ii) system node-specific information, and (iii) the relations between nodes and components (i.e., which components are running on which node). Architectures are described using an architecture description language (ADL). Initially all the ADL files are interpreted to generate the model that represents all the alternative architectures. The model of the system is maintained by the autonomic manager when (i) the application's architecture evolves[1] and (ii) when a new node is inserted or removed in the system. We suppose that all reconfiguration operations related to components go through the autonomic manager. Nodes are equipped

[1] e.g. When the system is initially deployed, when components are added or removed and when a component's configuration change.

with a discovery module that enables the autonomic manager to detect nodes insertion or removal.

4.2. Architecture comparison

With component-oriented technology that supports composition, a software system is structured as a hierarchy of interconnected components. As we discuss later, a component structure can be seen as a tree of components. Our third principle is to exploit the similarity between comparing component structures and trees. In particular, we choose to update an existing tree comparison algorithm into a component structure comparison algorithm, taking into account component semantics and identifying the changes we can do using dynamic reconfiguration operations.

4.3. Consistent change management

Change management aims at interpreting reconfiguration scripts while maintaining the system in a consistent state. As we see later, change management considers structural changes that can be applied on the distributed component structure. It can take the form of adding/removing components (via component factories) or modifying component attributes and bindings (via control interfaces seen in Section 6). Our fourth principle is that the autonomic manager must govern the lifecycle of the system's components for consistency reasons. This implies that components must provide reconfiguration-aware lifecycle operations.

4.4. Analysis and decision

The analysis and decision aspects of the control loop are not the main focus of this paper. They are realized using a rules engine based on a classical first-order predicate logic to perform inference. The inference rules are specified by means of a description language. We choose a rule system in order to support easily changing the management logic, and to avoid coupling this logic with low-level management code.

5. Architecture Overview

Our autonomic manager is designed for reusability. Administrators describe all the alternative architectures using an ADL (architecture description language) and add associated rules that will trigger specific architectures. These rules contain conditions that match against an event working set, which is dynamically updated by monitoring the system. Since many conditions can be true at a time, administrators typically have to specify priorities between architectures. The core of the autonomic manager (depicted in figure 1) is then composed of:

- The meta-data of the current architecture
- All the meta-data corresponding to the alternative architectures
- The event working set filled in real time by monitoring the runtime system.
- A rules engine that trigger architecture swaps according to the events that occur in of the system and the conditions associated with the architectures. Our system is based on the Java Rules Engine API described by the JSR-94 standard. Specifically, our prototype uses the Drools rules engine, which conforms to JSR-94 and provides a declarative language for defining rules.
- The architecture comparison tool used to compute the reconfiguration script in order to swap between the current running architecture and the target architecture requested by the rules engine.

The *Changes Manager* is used to interpret the reconfiguration script. This tool uses the *Nodes Manager* to allocate new, available nodes if necessary. It then contacts the *Software Resource Repository* to retrieve the necessary software resources and deploy them on the new nodes. In the following sections, we describe the component model we use, and then focus on architecture comparison and change management.

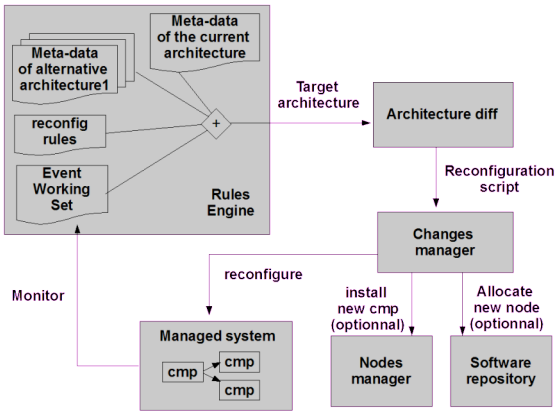

Figure 1. Overall architecture

6. The Fractal component model

The Fractal component model [8] is a general component model, which is intended to support implementing, deploying, and dynamically configuring, complex software systems, including in particular operating systems and middleware. This

motivates the three main features of the model: (i) support for composite components to provide a uniform view of applications at various levels of abstraction, (ii) introspection capabilities to discover the structure of a running system, and (iii) reconfiguration capabilities to deploy, and dynamically configure a system.

In more detail, a Fractal component is a run-time entity that is encapsulated and has one or more interfaces. An interface is an access point to a component supporting a finite set of methods. Interfaces can be of two kinds: server interfaces, which accept method calls, and client interfaces, which emit method calls. The signatures of both kinds of interface can be described by a standard Java interface declaration, with an additional role indication (server or client). A Fractal component can be composite, i.e., defined as an assembly of several sub-components, or primitive, i.e., encapsulating an executable program. Communication between Fractal components is only possible if their interfaces are bound. The Fractal model thus provides two mechanisms to define the architecture of an application: bindings between component interfaces, and encapsulation of components in a composite component.

The above features (hierarchical components, explicit bindings between components, separation between component interfaces and component implementation) are relatively classical. The originality of the Fractal model lies in its open reflective features. Indeed, Fractal components can be endowed with controllers, which provide access to a component internals and allowing component introspection and well-scoped dynamic reconfiguration. A controller provides a control interface and implements a control behavior for the component, such as controlling the activities in the components (suspend, resume). The Fractal model allows for arbitrary (including user defined) classes of controller. It specifies, however, several useful forms of controllers, which can be combined and extended to yield components with different control features. These controllers include the following:

- Attribute controller: supports an interface with getter and setter methods for attributes, that is, configurable component properties.
- Binding controller: supports binding and unbinding the client interfaces of the component to server interfaces.
- Content controller: supports listing, adding and removing the subcomponents of a composite component.
- Life-cycle controller: supports controlling the component execution (starting/stopping components). When a component is stopped, its state can be saved or restored.

Based on this component model, the meta-data associated with each component and maintained by our autonomic manager contain:

- The set of subcomponents that belong to the component (in the case of composite components)
- The set of client and server interfaces of the component
- The implementation of the functional interfaces of the component (in the case of primitive components)
- The set of attributes maintained by the component
- The set of binding involving component interfaces
- The machine where the component is running

7. Using a tree comparison algorithm

A software system built using a component technology such as Fractal is structured as a hierarchy of interconnected components[4]. Such component hierarchies represent the trees we want to compare. Finding the difference between two component systems is then analogous to the tree-to-tree comparison algorithms that have been used for finding differences between structured data such as LATEX files or XML [3][4]. These algorithms try to find a sequence of elementary operations which transform one tree into the other. Such sequences are called edit scripts and can be seen as reconfiguration scripts that update a tree into another. Selkow's algorithm [5] is a particular solution used when trees are of depth two. The algorithms developed by Tai [6] and Zhang[7] solve the generalization of the problem to trees of any depth. The difference is that tree transversal is performed in post-order with Zhang instead of pre-order. These algorithms compute the minimum set of operations (called the *minimum distance*) to update the first tree into the second one.

Our approach to comparing component trees relies on and extend Zhang's algorithm. Specifically, Zhang's algorithm works on simple labeled trees and assumes three kind of reconfiguration operations on a tree: adding/removing an element and modifying the label of an element. We update this algorithm to become aware of the component meta-model and of the available reconfiguration operations. Furthermore, Zhang's algorithm assumes that all reconfiguration operations can be applied on a tree without constraints, which is not true in our case. Our comparison

[4] We forbid component sharing

algorithm takes care of two specific problems: (i) the component identity and (ii) the difference between the expected and the possible kind of reconfiguration operations. We explain these problems in the following.

7.1. Identity

Comparing running component structures requires a method for recognizing components that are the 'same' in the two structures. For example, the tool must be able to differentiate between the migration of a component and the removal and the addition of different instances of the same component type. These choices are not equivalent. In the first case, the state of the instance must be preserved. In the second case, the new component is starting up and comes with a new state. As a result, we need a strong notion of component identity in the meta model. The identity function will be used by the algorithm to know if a component in the first tree T_1 is present in the second tree T_2. This mean that the two components are the same if they have the same identifier even though their configuration may differ. It also means that there is at least a sequence of operations that reconfigure the component instance in T_1 into the component instance in T_2. Our solution is to add a persistent component reference as meta-data in the component model. Components belonging to a given architecture are then referenced through a persistent id in the meta model, which is mapped to the current component's reference.

7.2. Expected and possible kind of changes

The kinds of changes we can expect derive from component semantics and cover the following:

- Modifying the structure of a component (adding/removing subcomponents, and modifying the interconnection pattern)
- Modifying the placement of components
- Modifying components' implementation (for primitive component)
- Modifying components' interfaces
- Modifying the set of attributes held by a component

The kinds of changes supported by our component model are the following:

- Adding/Removing components
- Adding/Removing bindings
- Getting/setting attributes

Thus, our model does not directly support changes related to interfaces, implementations and component placement; these changes require the use of compound reconfiguration operations that include deleting a component and re-creating it. The comparison algorithm must be able to detect these kinds of changes and the related compound reconfiguration operations. The algorithm computes what components have been deleted, what components have been added, what components have been reconfigured and what components have not been modified. It then generates the reconfiguration script to update the system. The reconfiguration script must ensure the consistency of the system. Each kind of reconfiguration operation has some constraints to maintain consistency. We will detail these issues in the next section and we present the compound operations and their associated constraints.

8. Change management

Performing arbitrary dynamic changes to an application may introduce inconsistencies and failures. Change management is concerned with maintaining consistency while minimizing the disruption of the running application [9]. Consistency is generally maintained whenever the new application version can resume computation from the former application global state. Our approach to maintaining consistency consists in associating a number of constraints with the reconfiguration operations and ensuring that the component structure satisfies these constraints during reconfiguration. Specifically, once the reconfiguration sequence has been computed, we consider the following constraints:

- Added components can be introduced in the system without any restrictions. All their attributes are new as well as their binding and subcomponent relationships.
- Removed components must be deleted with care from the system because other components may still have references to them. We therefore impose the following constraint: to remove a component, all bindings which point to the component should first be removed, and the component should then be stopped. Stopping the component ensures that (i) all its activities have been suspended, (ii) its state is consistent and can be serialized, and (iii) its activities can be resumed after the reconfiguration.
- Reconfigured components also require specific care. If these components have been reconfigured in terms of their attributes, bindings, or subcomponents, then they must be stopped before performing the changes. If the reconfiguration involves moving a component or modifying its interfaces, then

the following steps are followed: all bindings which point to the component should be removed, the component should be stopped, its state should be saved[5], the component should then be deleted and recreated on the target machine, its state and all the bindings involving the component should be restored, and finally its computation should be resumed. In the case of interface modifications, we also verify interface compliance with interconnected components.

9. Use Case

Our use case illustrates self-optimization, an autonomic behavior that maximizes server utilization with no human intervention. Figure 2 depicts a classical pattern in standard QoS infrastructures. In this pattern, a given server *S* is replicated at deployment time, and a front-end proxy *P* acts as a load balancer and distributes incoming requests among the replicas.

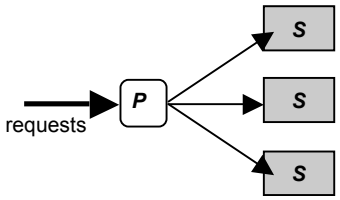

Figure 2. Load balancing among replicas

We want to provide a self-optimization manager which aims at autonomously increasing/decreasing the number of replicated servers used by the application when the load increases/decreases. To this purpose, the manager can swap between two architecture descriptions corresponding to two strategies: the first strategy is used when the load is low (i.e., under a specific threshold) and the second strategy is used when the load is high (i.e., above a threshold). The architecture corresponding to the first strategy is composed of one proxy component bound to three servers. The architecture corresponding to the second strategy is composed of one proxy component bound to ten servers. In this example, the server components are wrapped web containers[6]. The used sensor is a probe that collects CPU usage information on all the nodes where such a server is deployed. This probe computes a moving average of the collected data in order to remove artifacts characterizing the CPU consumption. To observe a general load indication of the whole replicated server, the autonomic manager calculates the average CPU load across all nodes. Actuators allow updating the server's configuration.

We illustrate in the following the ADL files corresponding to the alternative architectures and the conditions attached with each of the architectures. Figure 3 shows the ADL description for the first architecture, the low-load configuration, which contains a load balancer (PLB) bound to three web containers (tomcat). The autonomic manager uses this ADL description to initially deploy the system. The bundle tag specifies which software to install and the virtual node tag provides placement information: if two components have the same virtual node then they are placed on the same physical machine[7].

```
<component name="lowloadconfig">

<component name="plb" type="HTTPLB">
 <attributes>     name="port"     value="8081"
</attributes>
 <virtual-node name="node1" />
 <bundle> name="plb" version="plb" />
</component>

<component name="tomcat1" type="tomcat">
 <attributes>
  ...
  <attribute name="workerPort" value="8098" />
 </attributes>
 <virtual-node name="node2" />
 <bundle> name="tomcat" version="6.0" />
</component>

<component name="tomcat2" type="tomcat">
 <attributes>
  ...
  <attribute name="workerPort" value="8098" />
 </attributes>
 <virtual-node name="node3" />
 <bundle> name="tomcat" version="6.0" />
</component>

<component name="tomcat3" type="tomcat">
 <attributes>
  ...
  <attribute name="workerPort" value="8098" />
 </attributes>
 <virtual-node name="node4" />
 <bundle> name="tomcat" version="6.0" />
</component>

<binding
src="plb.workers"dest="tomcat1.endpoint" />
<binding
src="plb.workers"dest="tomcat2.endpoint" />
<binding
src="plb.workers"dest="tomcat3.endpoint" />
</component>
```

Figure 3. Low configuration

[5] When stopped, a component is serializable.

[6] We consider that web containers do not share their node with each others.

[7] The autonomic manager will allocate the real physicals machine.

Similarly, figure 4 shows the ADL description for the high-load configuration, in which the load balancer is bound to a pool of ten web containers.

```
<component name="highloadconfig">

<component name="plb" type="HTTPLB">
 <attributes>     name="port"     value="8081"
</attributes>
 <virtual-node name="node1" />
 <bundle> name="plb" version="plb" />
</component>

<pool name="tomcatCluster" begin="0" end="9"
<component name="tomcat1" type="tomcat">
 <attributes>
  <attribute name="workerPort" value="8098" />
 </attributes>
 <virtual-node name="node2" />
 <bundle> name="tomcat" version="6.0" />
</component>
</pool>

<binding       src="plb.workers"
       dest="tomcatCluster.endpoint" />
</component>
```

Figure 4. High configuration

We now describe the rules associated with these alternative configurations. These rules are very simple here, and they are described using Drools. Rules in Drools have a two-part structure (when <condition> then <action>) using first order logic for knowledge representation. Each time a new fact is inserted in the working memory, the inference engine matches facts against rules and infers conclusions that result in actions. We choose Drools for its expressiveness even though this limits the scalability of our prototype in terms of the number of facts and rules. Indeed, we currently target use cases with a limited scale.

```
rule "average load"
when
 $report : sensorReport()
 $avgload : from accumulate(
             LoadItem(report==$report,
                      $type==http,
                      $load : load )
                      average($load))
then
  # average load for all the nodes that
  # contain a web container is $load
  # delete the report
  # add the average load as a new fact
  modify(newload) {
    load=$avgload)
  }
end
```

According to the first rule ("average load"), the engine will initialize the load variable to zero for each sensorReport()[9] in the working memory. It will then iterate over all LoadItem() objects to calculate the average load into the $load variable. Only the load corresponding to nodes that contain a web container is taken into account. The engine will then modify the fact that contains the load in the working memory, which may trigger the two following rules:

```
rule "swap to lowloadconfig"
duration 10000
when
 $load : newload(load<lowthreshold) and
 currentarchi(type==highloadconfig)
then
  reconfigure(target==lowloadconfig)
end
```

The "swap to lowloadconfig" rule states that if the system is currently in the high load configuration and if the load is too low, then the system should be reconfigured into the low load configuration.

```
rule "swap to highloadconfig"
duration 10000
when
 $load : newload(load>highthreshold) and
 currentarchi(type==lowloadconfig)
then
 reconfigure(target==highloadconfig)
end
```

Similarly, the "swap to highloadconfig" rule concerns the transition to the low load configuration. To avoid unstable behavior and oscillation, the administrator can set a duration attribute for each rule. This attribute dictates that the rule will fire after the specified duration, if its condition is still true. In our example, the system will be reconfigured if the threshold condition remains valid for 10sec.

10. Conclusion

Autonomic computing has emerged as a promising approach to tackling the increasing complexity of our software systems. The approach relies on using autonomic managers that monitor a system and its environment, determine appropriate changes to the system in response to events, and execute these changes on the system. The contribution of this paper is to provide a generic tool to ease the development of autonomic managers. Using this tool, an administrator

[9] The report fact is periodically updated in the working memory by the system's sensor. This report contains an array list of the load for each node and a set of properties that indicate which kind of service run on the node. (e.g. : "http " means that a web container run on the given node).

provides a set of alternative architectures and specifies the conditions under which the architecture should be dynamically updated by the autonomic manager.

Our solution is based on system knowledge in the form of a set of interconnected components. This knowledge includes a model of the current system but also of the alternative architectures. A rule engine triggers reconfiguration, which uses an architecture diff tool together with a change manager to swap between system architectures. The architecture diff tool relies on a tree comparison algorithm that has been extended to take into account component model semantics. A first prototype has been implemented using Drools, and is currently being evaluated using the self-optimization scenario. Scalability and usability evaluation using additional self-management scenarios is postponed as a future work.

11. References

[1] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing", *IEEE Computer Magazine*, 36(1), 2003.

[2] S. Sicard, F. Boyer, Noel De Palma: Using components for architecture-based management: the self-repair case. ICSE 2008.

[3] D. T. Barnard, G. Clarke, and N. Duncan. Tree-to-tree correction for document trees. Technical report, Departement of Computing andf Information Science Queen's University Kingston Ontario, Canada, January 1995.

[4] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *18. International Conference on Data Engineering (ICDE) San Jose, California, USA, February 26-March 1, 2002*, 2002.

[5] S. M. Selkow. The tree-to-tree editing problem. Information Processing Letters, 6(6):184{186, December 1977.

[6] K.-C. Tai. The tree-to-tree correction problem.*Journal of the ACM*, 26(3):422–433, July 1979.

[7] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18:1245–1262, 1989.

[8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The Fractal Component Model and its Support in Java", *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, volume 36, number 11-12, 2006.

[9] J. Kramer, J. Magee, The evolving philosophers problem, IEEE Transactions on Software Engineering Volume 16, Issue 11, (November 1990) , Pages: 1293 - 1306 , Year of Publication: 1990, ISSN:0098-5589

[10] Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in Proceedings of the Twenty-seventh International Conference on Very Large Data Bases:Roma, Italy, 11–14th September, 2001, pages 581–590,Los Altos, CA 94022, USA, 2001. Morgan KaufmannPublishers.

[11] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-Diff: An effective change detection algorithm for XML documents. In 19th International Conference on Data Engineering, March 5 - March 8, 2003 – Bangalore, India, 2003.

[12] G. Valiente. "An efficient bottom-up distance between trees". Proceedings of the 8th International Symposium on String Processing and Information Retrieval, Santiago, Chile, November 13-15, 2001.

[13] "Concurrent Versions Systems". GNU CVS http://www.gnu.org/software/cvs

[14] R. Al-Ekram, A. Adma, O. Baysal. DiffX: An algorithm todetect changes in multi-version XML Documents., In IBM Centre for Advanced Studies Conference Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research,Toranto, Ontario, Canada, Pages: 1 – 11, 2005, ISSN:1705-7361.

[15] Z. Xing, E. Strouilia. UMLDiff: An algorithm for object-oriented design differencing.. In Automated Software Engineering in Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, Long Beach, CA, USA, Pages: 54 – 65, 2005, ISBN:1-59593